Dealing with Integer Overflow

GNAT Pro Offers Precise Control of Program Behavior

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

If an integer operation overflows and overflow checks are enabled, the Ada language definition allows the implementation to either raise an exception (Constraint_Error) or else — and in particular for intermediate results — to compute a correct mathematical result that is then used in subsequent operations. This flexibility may result in non-portable behavior, since mathematically associative operations such as integer ‘*’ are not necessarily associative at the machine level. The integer expression A*B*C can be evaluated as A*(B*C) or (A*B)*C, but one may be much more time consuming than the other depending on whether an overflow occurs.

Ada 2012’s contract-based programming features introduce some further considerations, since integer expressions in assertions such as ‘pre’ and ‘postconditions’ should generally be evaluated mathematically versus as machine integers. For example:

```ada
pre   A+B+C <= Integer'Last;
```

In a call such as P(Integer’Last, Integer’Last, Integer’Last) the precondition should evaluate as False rather than possibly raising an exception from an intermediate result overflow.

To address both the portability issue and the problem of mathematical versus machine interpretation of expressions in assertions, GNAT Pro provides comprehensive control of overflow behavior. The compiler can operate in three modes, with independent control over the treatment of expressions inside and outside assertions. The modes have the following effect when overflow checks are enabled:

- STRICT: all intermediate computations use the base type’s arithmetic, with a check for overflow outside the base type’s range.
- MINIAMID: most intermediate overflow checks are eliminated since computations are performed using the largest native integer type; and
- LIBERATE: no overflow occurs for intermediate results since computations are performed using arbitrary precision arithmetic as required.

The mode can be specified through either the OVERFLOW_CHECK pragma, or the ‘gnat’ or ‘gnat?’ compiler switch. For further details please refer to Annex D (Overflow Check Handling in GNAT) in the GNAT Pro User’s Guide.

CodePeer Selected by Saab EDS

Saab Electrical Defence Systems in Sweden has adopted the CodePeer static analysis tool for use on the GNAFFE project, a family of land and sea radar-based surveillance and air defense command and control systems. For this type of critical software, defects must be prevented at least detected and corrected before they get into deployed code. CodePeer has demonstrated its ability to find potential run-time and logic errors statically, significantly reducing design and testing requirements. Employing this tool during new software development helps the Saab engineers to avoid problematic constructs and to produce more robust code.

AdaCore is exhibiting at many of the conferences below. For up-to-date information on conferences where AdaCore is participating, please visit www.adacore.com/category/press-center/events/

**Conferences / Events**

**November 2012 – June 2013**

- **Automotive 2012**
  - November 14–15, 2012 / Karlsruhe, Germany
  - AdaCore is a sponsor, and Johannes Karg is presenting a tutorial on Formal Verification and “Test.”
  - www.automotive2012.de/

- **HILT 2012: High Integrity Language Technology**
  - December 2–4, 2012 / Boston MA, USA
  - AdaCore’s participation in the event is described in a companion article in this issue.
  - www.imagga.org/conference/hilt2012/

- **Paris Spaceweek**
  - December 17–19, 2012 / Paris, France
  - AdaCore is exhibiting at this event.
  - www.paris-spaceweek.com

- **RTTEC—Real Time & Embedded Computing Conference**
  - March 19, 2013 / Dallas TX, USA
  - AdaCore is exhibiting at this event.
  - www.rttec.com/conferences/view/113

- **Ada Europe 2013**
  - October 10–13, 2013 / Berlin, Germany
  - AdaCore is a sponsor of this conference.
  - www.ada-europe.org/2013

- **Design West**
  - March 19, 2013 / San Jose CA, USA
  - AdaCore is exhibiting at this conference (booth #790), and there are presentations from Ben Brogoli (Object-Oriented Programming for High-Level Systems: Piffits and How to Avoid Them) and Tucker Taft (Software Engineering in the Distributed, MultiCore World with Go, Rust, and Parasail).
  - Contessa-safetyclub-idc/42

- **GPPRObuild 2.0 Enhances Support for Multi-Language Projects**
  - Easier Deployment, New Build Options
  - A major new release of GPPRObuild, the powerful but easy-to-use tool for automating the construction of complex multi-language software, brings a number of enhancements including a new ‘complement_to’ tool, support for “force” to aggregate hierarchies of libraries.
  - GPPRObuild is based around “project files”, text files using an Ada-like notation to specify software project properties relevant to generating an executable or performing static analysis. Typical properties include the locations of source and object files, which languages the source code is written in, file naming conventions, and required switches.
  - Project files can be created and edited directly by the developer, or more simply through interactive wizards in the GPS and GNAITools IDEs. Through a project file a single invocation of GPPRObuild can generate a complete system or submit the components to various project-aware static analysis tools.
  - Until the release of GPPRObuild 2.0, the installation of the components into their destination locations remained a manual step.

- **GPPRObuild 2.0**: a new installation tool in GPPRObuild 2.0 now automates this last step. It uses an enhanced project file syntax that can describe how the software system should be built.

Another innovation in GPPRObuild 2.0 is support for grouping an entire hierarchy of dependent libraries into one single library file, to simplify the delivery and maintenance of a software product, since keeping track of multiple interdependent library files and making sure they are consistent can prove difficult.

To satisfy this need, the notion of “aggregate library projects” has been introduced. For such a project, all the software components across multiple subprojects are combined into a single library that can then be easily distributed.

GPPRObuild 2.0 is compatible with GPPRO versions 3.16a and 7.1. It is available on most host operating systems. For more information, please visit www.adacore.com. To learn more, please contact info@adacore.com.

**GPPRObuild Awarded Prize at Formal Methods Symposium**

A prototype release of GPPRObuild, a formal verification tool based on Ada 2012 contracts, received honors in a competition held at the International Symposium on Formal Methods in Paris in August 2012. The contest required the teams to apply verification tools to three sample programs, with the goal of allowing the teams to compare approaches and to learn from each other. GPPRObuild received the “Distinguished user-assistance tool” feature prize for its proof of integration and run-time assertion checking.

**GPPRObuild 2.0**: a new installation tool in GPPRObuild 2.0 now automates this last step. It uses an enhanced project file syntax that can describe how the software system should be built.

Another innovation in GPPRObuild 2.0 is support for grouping an entire hierarchy of dependent libraries into one single library file, to simplify the delivery and maintenance of a software product, since keeping track of multiple interdependent library files and making sure they are consistent can prove difficult.

To satisfy this need, the notion of “aggregate library projects” has been introduced. For such a project, all the software components across multiple subprojects are combined into a single library that can then be easily distributed.

GPPRObuild 2.0 is compatible with GPPRO versions 3.16a and 7.1. It is available on most host operating systems. For more information, please visit www.adacore.com. To learn more, please contact info@adacore.com.
Spotlighting a GAP Member

Universidad Politècnica de Madrid (Spain)

Ada, LEGO, and Real-Time at the Technical University of Madrid

A course in real-time programming, including such classical topics as concurrency, real-time scheduling and schedulability analysis, and i/O device programming, is an essential part of a Computer Science curriculum. Professors Juan Antonio de la Puente and Juan Zamaro have been teaching such a course at UPM for many years, using Ada as the main programming language. They chose Ada because of its comprehensive support for reliable software engineering and native concurrency. Formulating exercises and projects that are both realistic and appropriate for student use presents a challenge. The UPM professors have decided to base the labs on the minimal and low-cost LEGO MINDSTORMS NXT robotics platform, using a GNAT cross-development environment to build the Ada applications.

Peter Bradley, a Master’s Degree student in the Real-Time Systems and Telematic Services Architecture Research Group at UPM, has ported GNAT to LEGO MINDSTORMS NXT to a Linux host and has put together a set of software development tools for generating Ada real-time control systems that comply with the Ravenscar Profile on the LEGO robotics platform. These tools will be introduced in the real-time systems course as a basis for student projects, supporting all critical concurrency and real-time scheduling aspects, while offering compatibility with modeling tools such as Simulink. Students will thus be able to implement an embedded system from beginning to end. The simplicity of the Ravenscar Profile and the use of programming patterns for high-confidence software development is a big part of my focus, and it made good financial and technical sense at that point for us to join forces with AdaCore. At around that time I became interested in multicore programming, which led to an urgent need to design a new language, this time from scratch. This new language, ParaSail, is now a big part of my focus, and it is acting as a kind of testbed for new parallel programming language ideas that might find their way into AdaCore commercial products at some point. All of which goes well with my official AdaCore title of “Vice President and Director of Language Research.”

As the lead designer of Ada 95 and a major contributor to the subsequent language revisions, you have played a significant role in Ada’s evolution for more than twenty years. Any “lessons learned” from your experience?

Good language design requires a very firm shared vision about the mission of the language. This can be provided by a strong technical leader, or by a strong technical team, or by a strong technical community. As the number of cores/chips increases, and the price of computing in the “cloud” drops, I believe all developers will need to take advantage of these new resources. Unlike the exponential growth in clock speed we enjoyed for most of my career, this new exponential growth in parallel and distributed computing resources requires a significant paradigm shift for software development. I believe our languages need to change fundamentally to allow us to tap the full potential of this shift. As far as executable modeling, we are beginning to see the progression towards very high level, domain-specific languages; this trend was predicted decades ago but never quite came to pass. AdaCore is already a leader in the application of formal methods to programming with our SPARK Pro, CodePeer, and Ada 2012 offerings, all of which are bringing concepts such as pre- and postconditions to a much broader audience. I see such formal concepts becoming a standard part of industrial-strength software development, as society becomes more and more dependent on software for its critical infrastructure.

Tell us about your background and how you came to be involved with Ada and AdaCore. What is your current role?

I first learned about Ada (then under development) in the late 1970s when I was the systems programmer at the Harvard student computing center. I was already very interested in language design and implementation, and after joining Intermetrics in 1980 I first worked on one of the early Ada environments for the Air Force and then led the design team for the Ada 95 project. This project brought together some of the world’s top talent in programming languages and gave me the chance to advance a number of ideas that I had been thinking about for some time. Adding major features like Object-Oriented Programming and protected types to Ada 83 was a definite challenge, and there were some fireworks along the way, but amazingly by the end we were all still talking with one another and were quite proud of the final result. Fast forward seven years from 1996. We were the Internet bubble had burst, and Intermetrics (then called Avant!Com) had closed down. I went off to found SoCcheck along with several colleagues, basing the business on some intellectual property we had purchased from Avant!Com. This included the early technology underlying what would eventually become CodePeer.

Forward fast another nine years, to 2011. SoCcheck’s static analysis technology had matured and had been licensed to AdaCore for the CodePeer product, and it made good financial and technical sense at that point for us to join forces with AdaCore. All of which went well with my official AdaCore title of “Vice President and Director of Language Research.”

What do you see as the main directions for programming language design and research in the future?

The three main directions I see are parallel programming (both tightly coupled and loosely coupled), executable modeling languages, and formal methods. As the number of cores/chips increases, and the price of computing in the “cloud” drops, I believe all developers will need to take advantage of these new resources. Unlike the exponential growth in clock speed we enjoyed for most of my career, this new exponential growth in parallel and distributed computing resources requires a significant paradigm shift for software development. I believe our languages need to change fundamentally to allow us to tap the full potential of this shift. As far as executable modeling, we are beginning to see the progression towards very high level, domain-specific languages; this trend was predicted decades ago but never quite came to pass. AdaCore is already a leader in the application of formal methods to programming with our SPARK Pro, CodePeer, and Ada 2012 offerings, all of which are bringing concepts such as pre- and postconditions to a much broader audience. I see such formal concepts becoming a standard part of industrial-strength software development, as society becomes more and more dependent on software for its critical infrastructure.

Any hobbies or outside interests that you’d like to share?

I am a member of a masters track and field club, called “Moss Velocity,” and have been on the national men’s 50+ gold-medal 4x400m relay team a couple of times. Quite a thrill, and it keeps me in better shape than I have been at any earlier time in my life.

As the lead designer of Ada 95 and a major contributor to the subsequent language revisions, you have played a significant role in Ada’s evolution for more than twenty years. Any “lessons learned” from your experience?

Good language design requires a very firm shared vision about the mission of the language. This can be provided by a strong technical leader, or by a strong technical team, or by a strong technical community. As the number of cores/chips increases, and the price of computing in the “cloud” drops, I believe all developers will need to take advantage of these new resources. Unlike the exponential growth in clock speed we enjoyed for most of my career, this new exponential growth in parallel and distributed computing resources requires a significant paradigm shift for software development. I believe our languages need to change fundamentally to allow us to tap the full potential of this shift. As far as executable modeling, we are beginning to see the progression towards very high level, domain-specific languages; this trend was predicted decades ago but never quite came to pass. AdaCore is already a leader in the application of formal methods to programming with our SPARK Pro, CodePeer, and Ada 2012 offerings, all of which are bringing concepts such as pre- and postconditions to a much broader audience. I see such formal concepts becoming a standard part of industrial-strength software development, as society becomes more and more dependent on software for its critical infrastructure.
NXT to a Linux host and has put together a set of software development tools for generating Ada real-time control systems that comply with the Ravenscar Profile on the versatile and low-cost LEGO MINDSTORMS NXT robotics platform, using a GNAT cross development environment to build the Ada applications.

Formulating exercises and projects that are both realistic and appropriate for student use presents a challenge. The UPM professors have decided to base the labs on the

SPARK Pro 11.0 is a major release with many new features:
- Improved support for generic subprograms, including proofs for both absence of run-time errors and demonstration of partial correctness properties.
- Major improvements to the uses of function proofs, enabling developers to eliminate the vast majority of axiomatic Simplifier user rules.
- The addition of a new statement, the “~ assertion”, to support proofs.
- The promotion of SPARKBridge to a fully supported feature. This tool can be used to determine why the Simplifier or alternative provers cannot prove an undischarged verification condition.
- Improved support for partial views on run-time objects.
- Additional patterns/errors for automatic endianness conversion (‘Scalar_Storage_Order), and support for protected objects.
- Dimensionality checking with new aspects and packages.
- Initial integration into GNATbench.

The GNATbench 2.7 release represents a major upgrade. In addition to supporting both Eclipse 3.8 and 4.2, the new version provides an integration with CodePeer, supports mixed Ada-Java development for both native and Android projects, and directly supports GPRbuild.

Eclipse 3.8 and 4.2, the new version provides an integration with CodePeer, supports mixed Ada-Java development for both native and Android projects, and directly supports GPRbuild.

As the lead designer of Ada 95 and a major contributor to the subsequent language revisions, you have played a significant role in Ada’s evolution for more than twenty years. Any “lessons learned” from your experience are different. Almost everything we did was aimed toward making that capability elegant, powerful, and internally consistent.

As the lead designer of Ada 95 and a major contributor to the subsequent language revisions, you have played a significant role in Ada’s evolution for more than twenty years. Any “lessons learned” from your experience are different.

AdaCore at HI2012

What do you see as the main directions for programming language design and research in the future?

As the lead designer of Ada 95 and a major contributor to the subsequent language revisions, you have played a significant role in Ada’s evolution for more than twenty years. Any “lessons learned” from your experience are different.

Good language design requires a very firm shared vision about the mission of the language. This can be provided by a strong technical leader, or it can be done in a more ad hoc fashion, with some people more interested in language design and implementation, and others more interested in software development. I believe our languages need to change fundamentally to allow us to tap the full potential of this shift. As far as executable modeling, we are beginning to see the progress toward very high level, domain-specific languages; this trend was predicted decades ago but never quite came to pass.

As the lead designer of Ada 95 and a major contributor to the subsequent language revisions, you have played a significant role in Ada’s evolution for more than twenty years. Any “lessons learned” from your experience are different.

AIChE SIGAda’s annual international conference, renamed HILT (High Integrity Language Technology) to reflect its broader reach, was held in Boston during December 2-6. AdaCore was a Platinum Level sponsor and company personnel participated actively in the conference organization and program. Ben Bregoli was the Conference Chair, Tucker Taft was Program Co-Chair, and Greg Gica was Publicity Chair. Johannes Kanig, Hristian Kirtchev, Vincent Pucci, and Ed Schonberg, co-authors from AdaCore included Gebert Bosch, Claire Dross, Johannes Kanig, Hristian Kirtchev, Vincent Pucci, and Ed Schonberg.

Any hobbies or outside interests that you’d like to share?

I am a member of a masters track and field club, called “Moss Velocity,” and have been on the nation’s men’s 50+ gold-medal 4x400m relay team a couple of times. Quite a thrill, and it keeps me in better shape than I have been at any earlier time in my life.

Adapted from a conversation with Tucker Taft, Vice President, Director of Language Research, AdaCore US.
Dealing with Integer Overflow
GNAT Pro Offers Precise Control of Program Behavior

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

If an integer operation overflows and overflow checks are enabled, the Ada language definition allows the implementation to either raise an exception (Constraint_Error) or else — and in particular for intermediate results when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

For example:

\[ A + (B + C) \]

or

\[ A + B + C \]

Both these expressions are expressions that result in a correct result while the other one produces an overflow.

Ada 2013's contract-based programming features introduce some further considerations, since integer expressions like:

\[ A + B + C \]

must be prevented or at least detected and corrected before they get into deployed code, without losing any of Ada's ability to find potential run-time and logic errors statically.

This has been designed to meet the design of Ada 2013.

The implementation to either raise an exception (Constraint_Error) or else — and in particular for intermediate results when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

For example:

\[ A + (B + C) \]

or

\[ A + B + C \]

Both these expressions are expressions that result in a correct result while the other one produces an overflow.

Ada 2013's contract-based programming features introduce some further considerations, since integer expressions like:

\[ A + B + C \]

must be prevented or at least detected and corrected before they get into deployed code, without losing any of Ada's ability to find potential run-time and logic errors statically.

This has been designed to meet the design of Ada 2013.

The implementation to either raise an exception (Constraint_Error) or else — and in particular for intermediate results when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.

The discrepancy between the infinite set of mathematical integers and the finite set of integer values on any particular hardware has always presented a challenge for programming languages. A key issue is the semantics when an integer operation overflows, yielding a magnitude that exceeds the capacity of a machine integer.