

IML Konzepte

Gerrit Schulz
Universität Stuttgart
Institut für Softwaretechnologie
Breitwiesenstraße 20-22
D-70565 Stuttgart
schulzgt@studi.informatik.uni-stuttgart.de

Die Intermediate Language (IML) ist eine Zwischendarstellung für Reengineering Projekte. Sie wurde im Rahmen des Bauhaus Projekts an der Universität Stuttgart entwickelt und dient als Grundlage für die Analysen der Bauhaus Suite. Obwohl sie von den gängigen Zwischendarstellungen aus dem Compilerbau abgeleitet ist, weist sie zu diesen erhebliche Unterschiede auf.

Da diese Zwischendarstellungen im Allgemeinen auf einem annotierten abstrakten Syntaxbaum aufbauen und auch die IML-Darstellung einen Graphen mit Knoten und Kanten umfaßt, wird im Folgenden häufig auch vom „IML-Graphen“ die Rede sein.

1. Compilerbau und die IML

Im Compilerbau werden Zwischendarstellungen üblicherweise als Vorstufe zur Codegenerierung verwendet. Angesiedelt im Middle-End sollen sie durch eine einfachere Darstellung und z.B. architekturunabhängige Optimierungen die Aufgaben des Back-Ends erleichtern.

Die verwendeten Techniken zur Generierung und auch der Aufbau der IML ähnelt in weiten Teilen den Zwischendarstellungen aus dem Compilerbau. Es gibt jedoch auch einige signifikante Unterschiede zwischen den Anforderungen in Compilern und im Reengineering, die die Erstellung einer eigenen Zwischendarstellung für das Bauhaus Projekt unumgänglich machen.

In erster Linie ist es der fehlende Rückbezug auf den Quelltext, der die Compilerbau-Zwischendarstellungen für Reengineering Projekte ungeeignet macht. Kommen dann eventuell noch semantisch äquivalente Umformungen hinzu, geht zwischen Quelltext und Zwischendarstellung auch noch der syntaktische Zusammenhang verloren.

2. Die Designziele der Bauhaus IML

Oberstes Ziel der IML ist es, eine quelltextnahe Darstellung zu bieten, die trotzdem ein hohes Abstraktionsniveau zuläßt. Die Generierung von ausführbarem Maschinencode ist dabei sekundär, wenn gleich theoretisch möglich.

Quelltextnahe Darstellung bedeutet dabei im Wesentlichen die genaue Abbildung der Programmsyntax und einen direkten Rückverweis auf die Quellcodeposition. Um zusätzlich auch die Ausführungssemantik der Programme modellieren zu können, sind für manche Quelltextkonstrukte auch semantische Darstellungen vorhanden. Diese sind dann redundant zur syntaktischen Darstellung vorhanden oder es existiert keine Repräsentation im Quelltext wie z.B. bei impliziten Typumwandlungen.

Prinzipiell ist die IML sprachunabhängig, denn auch wenn im Augenblick nur C und Java verfügbar sind, befindet sich doch C++ in Entwicklung und Ada ist in Planung.

Ein weiteres Merkmal der IML ist die Bereitstellung der Infrastruktur für spätere Analysen und Darstellungen wie die Generierung der Static Single Assignment Form (SSA).

3. Die Generierung der IML

Die Generierung der IML erfolgt mit dem Tool CAFE, dem C Analyser Front-End, das die klassischen Funktionen eines Compiler Front-Ends implementiert. Dazu gehören eine Syntaxanalyse und semantische Analysen zur Namensbindung, Typauflösung und Bestimmung der Typgrößen.

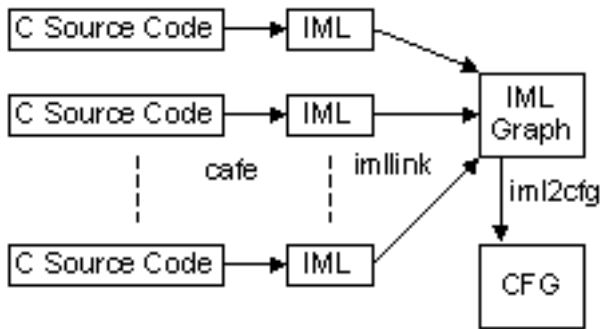


Abb. 1. IML Generierung

Um den kompletten IML-Graphen eines Programms zu erstellen, übersetzt man zunächst mit CAFE die einzelnen Quelldateien in IML-Dateien. Mit IMMLINK werden dann mehrere dieser Dateien zum IML-Graphen gebunden. Die IML Generierung orientiert sich also stark an der gewöhnlichen Übersetzung von C-Programmen.

Der erzeugte IML-Graphen läßt sich nun mit dem Tool IML2CFG um den Kontrollflußgraphen erweitern, der die sogenannten „Basic Blocks“ enthält. Weitere mögliche Schritte sind eine Pointer-Analyse und die Generierung der SSA-Form. Die entsprechenden Programme werden in der Bauhaus Dokumentation beschrieben. Zu beachten ist, daß alle diese Programme den bestehenden IML-Graphen erweitern und keinen neuen Graphen generieren. Die einzige Ausnahme bildet das Tool IML2RFG das den vom IML-Graphen völlig unabhängigen Ressourcen-Fluß-Graphen (RFG) erzeugt.

4. Das IML Klassenkonzept

Man hat sich beim Design der IML für ein hierarchisches Klassenkonzept entschieden, das sich an den gängigen objektorientierten Notationen orientiert. Es gibt Klassen mit Attributen und Methoden, abstrakte Klassen, die nicht direkt im Graphen auftauchen, und Interfaces. Interfaces sind ähnlich wie in Java aufgebaut, können aber im Gegensatz zu Java neben Methoden auch Attribute besitzen.

Die Knoten des IML-Graphen sind explizit als Objekte vorhanden, die Kanten existieren implizit über Pointerstrukturen.

Um ein flexibles und erweiterungsfähiges System zu erhalten, wurde ein Generator entwickelt, der aus einer abstrakten Darstellung Ada-Quelltext generiert. Auf diese Weise wird der gesamte Quelltext für die Basis-IML erzeugt. Die weiteren Schichten wie z. B. der CFG sind nicht mehr generiert, sondern werden von Hand erstellt. Im CFG kommt jedoch als einzige Erweiterung die Klasse „Basic_Block“ hinzu.

Als Basisklasse für die generierten Teile der IML dient die abstrakte Klasse „IML_Root“.

Der IML-Graph gliedert sich in:

- den Hierarchischen Programm Graph (HPG)
- Typen
- Objekte

4.1. Der Hierarchische Programm Graph (HPG)

Der hierarchische Programm Graph entspricht weitgehend dem abstrakten Syntaxbaum. In ihm sind die ausführbaren Teile des Programms enthalten, d.h. sämtliche Anweisungen, Ausdrücke und Unterprogramme. Dieser Teil ist es auch, der später von den Analysealgorithmen traversiert wird.

In der Klassenhierarchie ist der HPG mit der Basisklasse „HPGNode“ direkt unterhalb der Klasse „IML_Root“ angeordnet.

4.2. Typen und Objekte

Neben dem abstrakten Syntaxbaum wird von einem Compiler Front-End auch eine Symboltabelle angelegt, in der alle Bezeichner wie Variablen, Konstanten oder die Namen von Unterprogrammen gespeichert werden. Diese Informationen werden vor allem zur semantischen Analyse benötigt.

Die Integration dieser Informationen erfolgt nun nicht in Form einer Tabelle, sondern es werden Knoten mit der Basisklasse „SymNode“ in den IML-Graphen aufgenommen. Unterhalb dieser Klasse werden Typinformationen („T_Node“), wie z.B. Basisdatentypen oder Arrays, und Deklarationen („O_Node“) unterschieden. Zu den auch als „Objekte“ bezeichneten Deklarationen gehören Variablen, Konstanten oder Unterprogramme.

Bezüge auf Variablen oder Unterprogramme aus dem hierarchischen Programm Graphen erfolgen als semantische Kanten auf Knoten vom Typ „O_Node“.

4.3. Die IML Klassenhierarchie

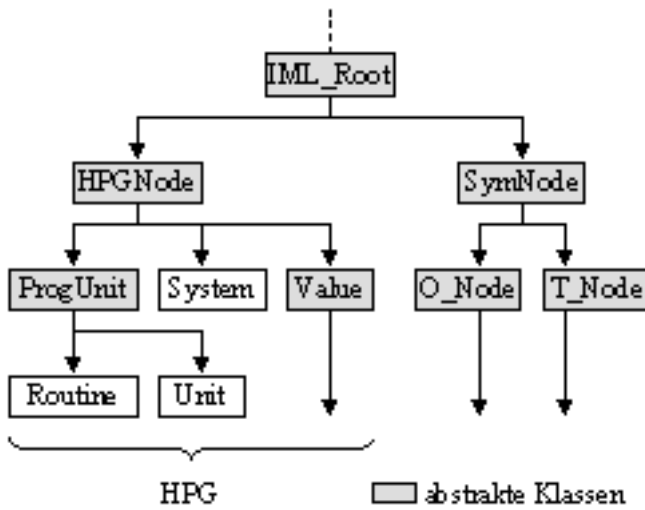


Abb. 2. IML Klassenhierarchie

Abb. 2 zeigt die Anordnung des HPG und der Symbolinformationen innerhalb der IML Klassenhierarchie. Alle weiteren Klassen des HPG werden von „Value“ abgeleitet. Oberhalb der Klasse „IML_Root“ existieren noch einige weitere Schichten, die im Zusammenhang mit nicht generiertem Code, wie z.B. den „Basic Blocks“ eine Rolle spielen. Eine jeweils aktuelle Übersicht über die IML Klassenhierarchie findet sich auf den Webseiten der Abteilung Programmiersprachen an der Universität Stuttgart.

4.4. Künstliche Knoten

Es ist anfangs schon erwähnt worden, daß es im Zusammenhang mit der Modellierung der Ausführungssemantik Knoten ohne syntaktische Repräsentation im Quelltext gibt. Dies sind sogenannte „künstliche Knoten“ die durch das „Artificial-Flag“ gekennzeichnet sind. Beispiele für „künstliche Knoten“ sind

- Read-Knoten beim Lesezugriff auf Variablen
- Implizite Typumwandlungen
- Initialisierungen von Variablen durch den Compiler

5. IML Implementierung

Die IML Implementierung stützt sich auf Ada tagged records, die folgende Formen von Attributen enthalten können:

- Pointer mit fester Zielklasse (Kanten)
- Listen von Pointern (Kanten mit Reihenfolge)

- Mengen von Pointern (Kanten ohne Reihenfolge)
- Builtins, nicht generierter Code (z.B. Ada Basistypen)
- Source Location (spezielles Builtin)

Hier wird auch die Realisierung von Kanten über typisierte Pointer deutlich. Bevor auf die einzelnen Attribute näher eingegangen wird, sei noch die Möglichkeit erwähnt, temporäre Attribute hinzuzufügen. Diese Attribute sind dann global für jeden Knoten verfügbar und können z.B. zur Speicherung von Zwischenergebnissen bei Analysen verwendet werden.

5.1. Syntaktische und semantische Kanten

In der IML werden zwei Arten von Kanten unterschieden. Syntaktische Kanten bilden die Programmstruktur und kommen nur im hierarchischen Programm Graph vor. Sie bilden somit den abstrakten Syntaxbaum des Programms. Semantische Kanten werden sehr vielfältig eingesetzt und dienen z.B. zum Verweis auf Variablen aus dem HPG oder zur Typprüfung. Zu jeder syntaktischen Kante wird automatisch eine semantische Kante in gegensätzlicher Richtung erzeugt, die sogenannte „Parent-Kante“. Auch in den Analysen werden ausschließlich semantische Kanten erzeugt. Eine Unterscheidung zwischen syntaktischen und semantischen Kanten erfolgt einzig über den Namen des Pointers. Die Implementierung ist äquivalent.

5.2. Builtins

Builtins werden verwendet, um von Hand erstellten Code in den Generator zu integrieren. Builtins erlauben es aus der Generatorsprache auf Datentypen zu verweisen, die anderweitig deklariert sind. Diese Datentypen können einfache Ada Basistypen wie Integer, Unbounded String etc. sein, in denen Informationen zum Knoten, wie z.B. der Name, gespeichert werden. Ein Beispiel hierfür ist auch das schon erwähnte „Artificial-Flag“.

Es sind aber auch beliebig komplexe Strukturen wie mehrdimensionale Arrays, Listen oder Hashtabellen möglich. Aus diesen Datenstrukturen sind über typisierte Pointer auch wieder Rückverweise auf den IML-Graphen möglich. Letztlich kann über Pointer auf tagged records die IML um Klassen erweitert, und Kanten auf Objekte dieser Klassen eingefügt werden. Gerade in den Erweiterungen der IML wird davon reger Gebrauch gemacht, als Beispiel sei die Klasse „Basic_Block“ erwähnt.

Eine gewisse Sonderstellung nimmt das Builtin „SLoc“, die Source Location ein. Dieses Builtin liefert den vielbeschworenen Rückbezug vom IML-Graphen auf den Quellcode. Die Source Location gibt genau an, in welcher Datei, Zeile und Spalte die entsprechende Quelltextrepräsentation

zu einem Knoten zu finden ist. Die Source Location ist allerdings kein eindeutiges Identifizierungsmerkmal eines Knotens da, im Zuge redundanter Darstellungen (Syntaktik, Semantik) und künstlicher Knoten gleiche Source Locations mehrfach auftreten können. Einige wenige Knoten besitzen auch keine Source Location wie z.B. die Klassen „System“ oder „Basic_Block“.

6. Zugriff auf den IML-Graphen

Nach der Betrachtung des IML Designs und der Implementation der Datenstrukturen stellt sich nun die Frage nach dem Zugriff auf den IML-Graphen. Dem objektorientierten Paradigma folgend gibt es für die einzelnen Klassen Methoden zum Zugriff auf die Attribute. Diese von der Bauhaus IML Bibliothek zur Verfügung gestellten Methoden, haben in etwa die folgende Form:

```
IML_Roots.Get_Artificial (
    Node : IML_Root);
```

Abfrage des „Artificial-Flags“

Diese Form des Zugriffs erfordert allerdings Kenntnisse darüber, welche Methoden für die einzelnen Klassen verfügbar sind und mit welcher Klasse man es aktuell zu tun hat. Für die bisherige Verwendung der IML, insbesondere für die Analysen, sind diese Voraussetzungen gegeben. Für das Studienprojekt „IML Browser“ entstehen durch diese Einschränkungen erhebliche Probleme. Da eine weitgehende Unabhängigkeit des „IML Browsers“ von der IML Bibliothek gefordert ist und auch spätere Änderungen der IML keine Änderungen am „IML Browser“ nach sich ziehen sollen, ist ein generischer Zugriff auf die Attribute erforderlich.

7. Ausblick auf Erweiterungen der IML

Im Bereich des automatisch generierten Codes sind Änderungen und Erweiterungen der IML recht komfortabel möglich. Neben Umbenennungen oder Änderungen von bestehenden Klassen ist auch die Erweiterung um neue Klassen ein ständiges Thema. Dabei wird auch angestrebt den Anteil des generierten Codes zu vergrößern. Änderungen sind auch für die Anpassung an objektorientierte Sprachen zu erwarten.

Im Rahmen des Studienprojekts „IML Browser“ ist neben der unbedingt notwendigen Implementierung eines generischen Zugriffs auf Attribute auch die Vergabe eindeutiger IDs für die Knoten wünschenswert. Zwar ist bei einem bereits geladenen IML-Graphen eine Knoten ID vorhanden, diese Information ist jedoch nicht persistent und ändert sich bei einem erneuten Laden des IML Graphen.

Zumindest dem ersten Punkt wird mit einem aktuell in der Entwicklung befindlichen neuen „Reflection Model“ Rechnung getragen.

8. Fazit

Die Bauhaus IML erfüllt die Anforderungen, die an eine für das Reengineering geeignete Zwischendarstellung gestellt werden. Im Zusammenspiel mit den Analysewerkzeugen der Bauhaus Suite bietet sie eine leistungsfähige Plattform für Reengineering Projekte. Durch den Einsatz eines Generators ist eine hohe Flexibilität gegeben und die Pflege und Erweiterung zumindest der Basis IML einfach möglich. Gerade die Erweiterung auf andere Sprachen und die Anpassungen an die Anforderungen des Studienprojekts „IML Browser“ werden für eine kontinuierliche Weiterentwicklung der IML sorgen.